AD-A245 550

ATION PAGE

Form Approved
OPM No. 0704-0188

2

1. AGENCY USE ONLY *(Leave Blank)*

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED
Final: 16 Nov 1990 to 01 Jun 1993

4. TITLE AND SUBTITLE
Rational, M68020/OS-2000 Cross-Development Facility, Version 7, R1000 Series 300(Host) to Philips PG2100 (Target), 901116W1.11081

5. FUNDING NUMBERS

6. AUTHOR(S)
Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB, Dayton, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-424-1290

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Rational, M68020/OS-2000 Cross-Development Facility, Version 7, Wright-Patterson, AFB, R1000 Series 300(Host) to Philips PG2100 (Target), ACVC 1.11.

92-03091

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 16 November 1990.

Compiler Name and Version:	M68020/OS-2000 Cross-Development Facility, Version 7
Host Computer System:	R1000 Series 300, Rational Environment, Version D_12_24_0
Target Computer System:	Philips PG2100, OS-2000 Release 2.0
Customer Agreement Number:	90-07-20-RAT

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901116W1.11081 is awarded to Rational. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

Steven P. Wilson

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

John P. Solomond

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

AVF Control Number: AVF-VSR-424-1290
19 November 1991
90-07-20-RAT

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901116W1.11081
Rational
M68020/OS-2000 Cross-Development Facility, Version 7
R1000 Series 300 => Philips PG2100

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 16 November 1990.

Compiler Name and Version:	M68020/OS-2000 Cross-Development Facility, Version 7
Host Computer System:	R1000 Series 300, Rational Environment, Version D_12_24_0
Target Computer System:	Philips PG2100, OS-2000 Release 2.0
Customer Agreement Number:	90-07-20-RAT

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901116W1.11081 is awarded to Rational. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

Steven P. Wilson

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

Customer:	Rational
Ada Validation Facility:	ASD/SCEL, Wright-Patterson AFB OH 45433-6503
ACVC Version:	1.11

Ada Implementation

Compiler Name:	M68020/OS-2000 Cross-Development Facility, Version 7
Host Architecture:	R1000 Series 300
Host Operating System:	Rational Environment Version D_12_24_0
Target Architecture:	Philips PG2100
Target Operating System:	OS-2000

Customer's Declaration

I, the undersigned, representing Rational, declare that Rational has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration. I declare that Rational is the owner of the above implementation and the certificates shall be awarded in the name of the owners corporate name.

Date: 9/27/90

David H. Bernstein
Vice President, Products Group

Rational
3320 Scott Blvd.
Santa Clara, CA 95054

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1 USE OF THIS VALIDATION SUMMARY REPORT 1-1
1.2 REFERENCES 1-2
1.3 ACVC TEST CLASSES 1-2
1.4 DEFINITION OF TERMS 1-3

CHAPTER 2 IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS 2-1
2.2 INAPPLICABLE TESTS 2-1
2.3 TEST MODIFICATIONS 2-4

CHAPTER 3 PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT 3-1
3.2 SUMMARY OF TEST RESULTS 3-1
3.3 TEST EXECUTION 3-2

APPENDIX A MACRO PARAMETERS

APPENDIX B COMPILATION SYSTEM OPTIONS

APPENDIX C APPENDIX F OF THE Ada STANDARD

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 12 October 1990.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
C74308A	B83022B	B83022H	B83025B	B83025D	B83026B
B85001L	C83026A	C83041A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE2117A	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35702A, C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45624A checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 5. For this implementation, MACHINE_OVERFLOWS is TRUE.

C45624B checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 6. For this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y checks for a predefined fixed-point type other than DURATION.

C96005B checks for values of type DURATION'BASE that are outside the range of DURATION. There are no such values for this implementation.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

CE2107B, CE2107D, C2107E, CE2107G, and CE2107L attempt to associate two internal files for writing with the same external file; this implementation raises USE_ERROR on the attempt to open the second file.

CE2111D and CE2111H attempt to reset one of two internal files that are associated with the same external file; this implementation raises USE_ERROR when the reset operation is attempted.

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A attempt to associate multiple internal files with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

CE3202A expects that function NAME can be applied to the standard input and output files; in this implementation these files have no names, and USE_ERROR is raised. (See section 2.3.)

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 99 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B22003B	B22004A	B22004B	B22004C	B23002A
B23004A	B23004B	B24001A	B24001B	B24001C	B24005A
B24005B	B24007A	B24009A	B24204B	B24204C	B24204D
B25002B	B26001A	B26002A	B26005A	B28003A	B28003C
B29001A	B2A003B	B2A003C	B2A003D	B2A007A	B32103A
B33201B	B33202B	B33203B	B33301A	B33301B	B35101A
B36002A	B37106A	B37205A	B37307B	B38003A	B38003B
B38009A	B38009B	B41201A	B44001A	B44004A	B44004B
B44004C	B44004D	B44004E	B45205A	B48002A	B48002D
B53003A	B55A01A	B56001A	B63001A	B63001B	B64001B
B64006A	B67001A	B67001B	B67001C	B67001D	B67001H
B71001A	B71001G	B71001M	B74003A	B74307B	B83E01C
B83E01D	B83E01E	B91001F	B91001H	B91003E	B95001D
B95003A	B95004A	B95006A	B95007B	B95079A	BA1001B
BB3005A	BC1109A	BC1109B	BC1109C	BC1109D	BC1303F
BC2001D	BC2001E	BC3003A	BC3003B	BC3005B	BC3013A
BE2210A	BE2413A	B51001A			

CE3202A was graded inapplicable by Evaluation Modification as directed

by the AVO. This test applies function NAME to the standard input file, which in this implementation has no name; USE_ERROR is raised but not handled, so the test is aborted. The AVO ruled that this behavior is acceptable pending any resolution of the issue by the ARG.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

David H. Bernstein
3320 Scott Blvd.
Santa Clara CA 95054

For a point of contact for sales information about this Ada implementation system, see:

David H. Bernstein
3320 Scott Blvd.
Santa Clara CA 95054

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordancc with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a)	Total Number of Applicable Tests	3795
b)	Total Number of Withdrawn Tests	81
c)	Processed Inapplicable Tests	93
d)	Non-Processed I/O Tests	0
e)	Non-Processed Floating-Point Precision Tests	201
f)	Total Number of Inapplicable Tests	294
g)	Total Number of Tests for ACVC 1.11	4170

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 294 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by FTP, and run. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

Option | Switch	Effect
Create_Subprogram_Specs = False	When a library unit subprogram body is added to the program library, do not automatically create a corresponding subprogram specification.
Linker_Command_File = "!VALIDATION.ACVC_1_11.MC68020_OS2000.MISCELLANY.MODIFIED_LINKER_COMMANDS"	Overrides the default linker command file with one that specifies inclusion of assembly language modules needed for pragma interface tests.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN—also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$MAX_IN_LEN	254
$ACC_SIZE	32
$ALIGNMENT	1
$COUNT_LAST	1000000000
$DEFAULT_MEM_SIZE	2147483647
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	MC68020_OS2000
$DELTA_DOC	0.0000000004656612873077392578125
$ENTRY_ADDRESS	SYSTEM.TO_ADDRESS (0)
$ENTRY_ADDRESS1	SYSTEM.TO_ADDRESS (0)
$ENTRY_ADDRESS2	SYSTEM.TO_ADDRESS (0)
$FIELD_LAST	2147483647
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	1.0
$GREATER_THAN_DURATION_BASE_LAST	131073.0
$GREATER_THAN_FLOAT_BASE_LAST	2.0E308
$GREATER_THAN_FLOAT_SAFE_LARGE	2#11111111111111111.1111111#E111

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E308
$HIGH_PRIORITY	255
$ILLEGAL_EXTERNAL_FILE_NAME1	BAD_CHARACTERS&<>=
$ILLEGAL_EXTERNAL_FILE_NAME2	CONTAINS_WILDCARDS*
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006F1.TST")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	ASM
$LESS_THAN_DURATION	-1.0
$LESS_THAN_DURATION_BASE_FIRST	-131073.0
$LINE_TERMINATOR	ASCII.CR
$LOW_PRIORITY	0
$MACHINE_CODE_STATEMENT	NULL;
$MACHINE_CODE_TYPE	NULL
$MANTISSA_DOC	31
$MAX_DIGITS	15
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2147483648
$MIN_INT	-2147483648

MACRO PARAMETERS

$NAME	SHORT_SHORT_INTEGER
$NAME_LIST	MC68020_OS2000
$NAME_SPECIFICATION1	X2120A
$NAME_SPECIFICATION2	X2120B
$NAME_SPECIFICATION3	X3119A
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2147483647
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	MC68020_OS2000
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	NEW INTEGER
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	8192
$TICK	1.22070312500000E-04
$VARIABLE_ADDRESS	SYSTEM.TO_ADDRESS (16#2380000#)
$VARIABLE_ADDRESS1	SYSTEM.TO_ADDRESS (16#2380040#)
$VARIABLE_ADDRESS2	SYSTEM.TO_ADDRESS (16#23800E0#)
$YOUR_PRAGMA	NICKNAME

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

```
PROCESSOR            SWITCH                        TYPE              VALUE

 Cross_Cg . Asm_Source                         : Boolean             := False
 -- Controls retention of assembly source code generated by the compiler.

 Cross_Cg . Auto_Download                      : Boolean             := True
 -- Controls whether the result of partially linking a main program is
 -- automatically downloaded to the target machine, using FTP switches to
 -- determine the destination.  Applies only to targets that have a final
 -- link step on the target machine.

 Directory . Create_Internal_Links             : Boolean             := True
 -- Controls whether internal links are created automatically when the
 -- visible parts of library units are created.  Internal links for library
 -- units are created in the set of links for the nearest enclosing world.
 -- The default is True.  The full switch name is
 -- Directory.Create_Internal_Links. (For further information on links, see
 -- the Key Concepts section of the Library Management (LM) Reference
 -- Manual.) .

* Directory . Create_Subprogram_Specs          : Boolean             := False
 -- Controls whether specifications for library-unit subprograms are created
 -- automatically.  The contents of these specifications are created the
 -- first time the body is successfully installed.  The "with" clause for
 -- the specification is derived from the "with" clauses in the body. Only
 -- those "with" clauses required to promote the specification are included.
 --  The default is True.  The full switch name is
 -- Directory.Create_Subprogram_Specs.

 Cross_Cg . Debugging_Level                    : Debug_Level         := Full
 -- Cross_Cg.Debugging_Level controls the amount of debugging assistance put
 -- into the object module when coding an Ada unit.
 --
```

```
-- The possible values are:
--    None    : (Default) No debugging information produced.
--    Partial : Debugging tables produced but optimizations are not
-- inhibited.
--    Full    : Debugging tables produced, and optimizations inhibited.
--
-- "Optimizations inhibited" means that code motion across statement
-- boundaries will not occur, and the lifetimes of variables will not be
-- reduced.

 Cross_Cg . Enable_Code_Pooling          : Boolean              := False
-- When true, optimizations which would prevent link time code pooling are
-- inhibited.  Link time code pooling is only attempted on units that were
-- compiled with this switch set to True.

 Cross_Cg . Inlining_Level               : Inlining_Level       := Inter_Uni
-- Cross_Cg.Inlining_Level determines how the compiler treats Pragma
-- Inline.
--
-- The possible values are:
--   None       : Ignore pragma Inline.
--   Intra_Unit : Honor pragma Inline within a compilation unit, but ignore
-- pragma Inline applied to subprograms in other compilation units (thus
-- avoiding introduction of additional compilation dependencies).
--   Inter_Unit : (Default) Honor all Inline pragmas.
--   Full       : Honor all Inline pragmas, and additionally perform
-- automatic inlining of small subprograms within a compilation unit.

 Cross_Cg . Linker_Command_File          : String               :=
  "!VALIDATION.ACVC1_11.MC68020_OS2000.MISCELLANY.MODIFIED_LINKER_COMMANDS"
-- Cross_Cg.Linker_Command_File overrides the default file name for the
-- linker command file.  The name of the linker command file is resolved in
-- the context of the current switch file.

 Cross_Cg . Linker_Cross_Reference       : Boolean              := False
-- Cross_Cg.Linker_Cross_Reference controls whether a cross-reference of
-- external symbols to modules being linked is produced in the link map.

 Cross_Cg . Linker_Eliminate_Dead_Code   : Boolean              := True
-- Cross_Cg.Linker_Eliminate_Dead_Code controls whether the linker removes
-- unreachable subprograms from the executable program image.

 Cross_Cg . Linker_Pool_Code             : Boolean              := False
-- Controls whether the linker eliminates redundant subprograms from the
-- executable program image. Redundant subprograms are those that are
-- reachable from the main program but whose code is identical to that of
-- some other subprogram in the program.  Only comp units that were
-- compiled with the switch Enable_Code_Pooling set to True are eligible
-- for code pooling at link time.

 Cross_Cg . Linker_Pool_Literals         : Boolean              := True
-- Controls whether the linker eliminates redundant literals from the
-- executable program image.
```

```
Cross_Cg . Listing                       : Boolean              := False
-- Controls generation of machine code listing file.

Cross_Cg . Optimization_Level            : Integer range 0 .. 3 := 3
-- Cross_Cg.Optimization_Level controls the amount of optimization
-- performed during code generation.
--
-- The possible values are:
--    0 : Minimal Optimization
--    1 : Unimplemented
--    2 : Unimplemented
--    3 : Maximal Optimization.

Directory . Require_Internal_Links       : Boolean              := True
-- Controls whether failure to create internal links (as controlled by the
-- Directory.Create_Internal_Links switch) generates an error.  The default
-- (True) is to treat the failure to generate links as an error and to
-- discontinue the operation.  If the Directory.Create_Internal_Links
-- switch is set to False, this switch has no effect.  The full switch name
-- is Directory.Require_Internal_Links.

Cross_Cg . Suppress_All_Checks           : Boolean              := False
-- When true, this switch has the same effect as a pragma Suppress_All at
-- the beginning of the each Ada unit in the library.

Cross_Cg . Target_Linker_Script          : String               := ""
-- Cross_Cg.Target_Linker_Script overrides the default file name for the
-- target linker script file.  This name of this file is resolved in the
-- context of the current switch file. This switch applies only to targets
-- having a final link step on the target machine.
```

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is
    ..........
    type Integer is range -2147483648 .. 2147483647;
    type Short_Short_Integer is range -128 .. 127;
    type Short_Integer is range -32768 .. 32767;

    type Float is digits 6 range -16#1.FFFF_FE# * 2.0 ** 127 ..
                                  16#1.FFFF_FE# * 2.0 ** 127;

    type Long_Float is digits 15 range
                    -16#1.FFFF_FFFF_FFFF_F# * 2.0 ** 1023 ..
                     16#1.FFFF_FFFF_FFFF_F# * 2.0 ** 1023;

    type Duration is delta 16#1.0# * 2.0 **-14
                        range -16#1.0# * 2.0 ** 17 ..
                               16#1.FFFF_FFFC# * 2.0 ** 16;
    ..........
end STANDARD;
```

Appendix V: Appendix F to the LRM for the Mc68020_Os2000 Target

The *Reference Manual for the Ada Programming Language* (LRM) specifies that certain features of the language are implementation-dependent. It requires that these implementation dependencies be defined in an appendix called Appendix F. This is Appendix F for the Mc68020_Os2000 target, compiler version 5. It contains materials on the following topics listed for inclusion by the LRM on page F-1:

- Implementation-dependent pragmas
- Implementation-dependent attributes
- Package System
- Representation clauses
- Implementation-dependent components
- Interpretation of expressions that appear in address clauses
- Unchecked conversion
- Implementation-dependent characteristics of I/O packages

These topics appear in section and subsection titles of this appendix. The appendix contains other topics mentioned in the LRM as being implementation-dependent. For these, a reference to the LRM is given in the section or subsection title. In addition, this appendix contains sections on predefined pragmas, predefined attributes, and size of objects. This material is included here because of implementation dependencies and close relationship to LRM-mandated topics.

IMPLEMENTATION-DEPENDENT PRAGMAS

The MC68020/OS-2000 cross-compiler supports pragmas for application software development in addition to those listed in Annex B of the LRM. They are described below, along with additional clarifications and restrictions for pragmas defined in Annex B of the LRM.

Pragma Main

A parameterless library-unit procedure without subunits can be designated as a main program by including a pragma Main at the end of the unit specification or body. This pragma causes the linker to run and create an executable program when the body of this subprogram is coded. Before a unit having a pragma Main can be coded, all units in the *with* closure of the unit must

be coded.

The pragma Main has three arguments:

- **Target**: A string specifying the target key. If this argument appears and it does not match the current target key, the pragma Main is ignored. If the Target parameter matches the current target key or does not appear, pragma Main is honored. A single source copy of a main program can be used for different targets by putting in multiple Main pragmas with different target parameters and different stack sizes and/or different heap sizes.
- **Stack_Size**: A static integer expression specifying the size in bytes of the main task stack. If not specified, the default value is 4096 (4K) bytes.
- **Heap_Size**: A static integer expression specifying the size in bytes of the heap. If not specified, the default value is 64K bytes.

The complete syntax for this pragma is:

```
pragma_main   ::=  PRAGMA MAIN
                   [ ( main_option { , main_option } ) ] ;

main_option   ::=  TARGET     => simple_name |
                   STACK_SIZE => static_integer_expression |
                   HEAP_SIZE  => static_integer_expression
```

The pragma Main must appear immediately after the declaration or body of a parameterless library-unit procedure without subunits.

Using the Target Parameter

Using the Target parameter forces the pragma to be ignored for all targets but the one specified. This enables joined views of a procedure to have different effects according to the target. One use is to avoid the effects of declaring a pragma Main when the target is Rational:

```
pragma Main (Target => Mc68020_Os2000);
```

Another use is to specify different stack or heap sizes for different targets. For example:

```
procedure Show_Pragma_Main is
begin
    Do_Something;
end Show_Pragma_Main;
pragma Main (Target => Mc68020_Os2000,         Heap_Size => <10*1024>);
pragma Main (Target => <another target key>, Heap_Size => <20*1024>);
```

The procedure Show_Pragma_Main will be a main program in both an Mc68020_Os2000 view and a view for the other target. The heap sizes for the two targets will be as specified by the different pragma Mains.

Multiple pragma Mains may be placed in the specification, the body, or both. If more than one pragma Main is specified with the same target parameter, only one of the pragmas will have any effect. The first pragma Main in the specification (if there is one) will be chosen; otherwise,

the first one in the body will be chosen.

Pragmas Import_Procedure and Import_Function

A subprogram written in another language (typically, assembly language) can be called from an Ada program if it is declared with a pragma Interface. The rules for placement of pragma Interface are given in Section 13.9 of the LRM. Every interfaced subprogram must have an importing pragma that is recognized by the MC68020/OS-2000 cross-compiler, either Import-_Procedure or Import_Function. These pragmas are used to declare the external name of the subprogram and the parameter-passing mechanism for the subprogram call. If an interfaced subprogram does not have an importing pragma, or if the importing pragma is incorrect, pragma Interface is ignored.

Importing pragmas can be applied only to nongeneric procedures and functions.

The pragmas Import_Procedure and Import_Function are used for importing subprograms. Import_Procedure is used to call a non-Ada procedure; Import_Function, a non-Ada function.

Each import pragma must be preceded by a pragma Interface; otherwise, the placement rules for these pragmas are identical to those of the pragma Interface.

The importing pragmas have the form:

```
importing_pragma   ::=  PRAGMA importing_type
                        ( [ INTERNAL => ] internal_name
                          [ , [ EXTERNAL => ] external_name ]
                          [ [ , [ PARAMETER_TYPES => ]
                                              parameter_types ]
                            [ , [ RESULT_TYPE => ] type_mark ] ]
                            [ , NICKNAME => string_literal ]
                          [ , [ MECHANISM => ] mechanisms ] ) ;

importing_type     ::=  IMPORT_PROCEDURE | IMPORT_FUNCTION |
                        IMPORT_VALUED_PROCEDURE
internal_name      ::=  identifier |
                        string_literal    -- An operator designator

external_name      ::=  string_literal

parameter_types    ::=  NULL | ( type_mark { , type_mark } )

mechanisms         ::=  mechanism_name |
                        ( mechanism_name { , mechanism_name } )

mechanism_name     ::=  VALUE | REFERENCE
```

The internal name is the Ada name of the subprogram being interfaced. If more than one subprogram is in the declarative region preceding the importing pragma, the correct subprogram must be identified by either using the argument types (and result type, if a function) or specifying the nickname. See pragma Nickname below.

The purpose of the Parameter_Types argument is to distinguish among two or more overloaded subprograms having the same internal name. The value of the Parameter_Types argument is a list of type or subtype names separated by commas and enclosed in parentheses. Each name corresponds positionally to a formal parameter in the subprogram's declaration. If the subprogram has no parameters, the list consists of the single word *null*. The Result_Type argument serves the same purpose for functions; its value is the type returned by the function.

The external designator, specified with the External parameter, is a character string that is an identifier suitable for the MC68020 assembler. If the external designator is not specified, the internal name is used.

The Mechanism argument is required if the subprogram has any parameters. The argument specifies, in a parenthesized list, the passing mechanism for each parameter to be passed. There must be a mechanism specified for each parameter listed in Parameter_Types and they must correspond positionally. The types of mechanism are as follows:

- Value: Specifies that the parameter is passed on the stack by value.
- Reference: Specifies that the address of the parameter is passed on the stack.

For functions, it is not possible to specify the passing mechanism of the function result; the standard Ada mechanism for the given type of the function result must be used by the interfaced subprogram. If there are parameters, and they all use the same passing mechanism, an alternate form for the Mechanism argument can be used: instead of a parenthesized list with an element for each parameter, the single mechanism name (not parenthesized) can be used.

Examples:

```
procedure Locate (Source: in String;
                  Target: in String;
                  Index:  out Natural);

pragma Interface (Assembler, Locate);
pragma Import_Procedure
         (Internal        => Locate,
          External        => "STR$LOCATE",
          Parameter_Types => (String, String, Natural),
          Mechanism       => (Reference, Reference, Value));


function Pwr (I: Integer; N: Integer) return Float;
function Pwr (F: Float; N: Integer) return Float;

pragma Interface (Assembler, Pwr);

pragma Import_Function
         (Internal        => Pwr,
          Parameter_Types => (Integer, Integer),
          Result_Type     => Float,
          Mechanism       => Value,
          External        => "MATH$PWR_OF_INTEGER");
```

```
pragma Import_Function
          (Internal        => Pwr,
           Parameter_Types => (Float, Integer),
           Result_Type     => Float,
           Mechanism       => Value,
           External        => "MATH$PWR_OF_FLOAT");
```

Pragmas Export_Procedure and Export_Function

A subprogram written in Ada can be made accessible to code written in another language by using an exporting pragma defined by the MC68020/OS-2000 cross-compiler. The effect of such a pragma is to give the subprogram a global symbolic name that the linker can use when resolving references between object modules.

Exporting pragmas can be applied only to nongeneric procedures and functions.

Exporting a subprogram does not export the mechanism used by the compiler to perform elaboaration checks; calls from other languages to an exported subprogram whose body is not yet elaboarated may have unpredictable results when the subprogram body references objects that are not yet elaborated. Elaboration checks within the Ada program are not affected by the exporting pragma.

An exporting pragma can be given only for a subprogram that is a library unit or that is declared in the specification of a library package. An exporting pragma can be placed after a subprogram body only if the subprogram does not have a separate specification. Thus, an exporting pragma cannot be applied to the body of a library subprogram that has a separate specification.

These pragmas have arguments similar to those of the importing pragmas, except that it is not possible to specify the parameter-passing mechanism. The standard Ada parameter-passing mechanisms are chosen. For descriptions of the pragma's arguments (Internal, External, Parameter_Types, Result_Type, and Nickname), see the preceding section on the importing pragmas.

The full syntax of the pragmas for exporting subprograms is:

```
exporting_pragma  ::=  PRAGMA exporting_type
                       ( [ INTERNAL => ] internal_name
                         [ , [ EXTERNAL => ] external_name ]
                         [ [ , [ PARAMETER_TYPES => ] parameter_types ]
                           [ , [ RESULT_TYPE => ] type_mark ] |
                           [ , NICKNAME => string_literal ] ] ) ;
exporting_type    ::=  EXPORT_PROCEDURE | EXPORT_FUNCTION
internal_name     ::=  identifier |
                       string_literal   -- An operator designator
external_name     ::=  string_literal
parameter_types   ::=  NULL | ( type_mark { , type_mark } )
```

Examples:

```
procedure Matrix_Multiply(A, B: in Matrix; C: out Matrix);
pragma Export_Procedure (Matrix_Multiply);
-- External name is the string "Matrix_Multiply"
function Sin (R: Radians) return Float;
pragma Export_Function
           (Internal => Sin,
            External => "SIN_RADIANS");
-- External name is the string "SIN_RADIANS"
```

Pragma Export_Elaboration_Procedure

The pragma Export_Elaboration_Procedure makes the elaboration procedure for a given compilation unit available to external code by defining a global symbolic name. This procedure is otherwise unnamable by the user. Its use is confined to the exceptional circumstances where an Ada module is not elaborated because it is not in the closure of the main program or the main program is not an Ada program. This pragma is not recommended for use in application programs unless the user has a thorough understanding of elaboration, runtime, and storage model considerations.

The pragma Export_Elaboration_Procedure must appear immediately following the compilation unit.

The complete syntax for this pragma is:

```
pragma_export_elaboration_procedure ::=
    PRAGMA EXPORT_ELABORATION_PROCEDURE ( EXTERNAL_NAME => external_name );

external_name ::= string_literal
```

Pragmas Import_Object and Export_Object

Objects can be imported or exported from an Ada unit with the pragmas Import_Object and Export_Object. The pragma Import_Object causes an Ada name to reference storage declared and allocated in some external (non-Ada) object module. The pragma Export_Object provides an object declared within an Ada unit with an external symbolic name that the linker can use to allow another program to access the object. It is the responsibility of the programmer to ensure that the internal structure of the object and the assumptions made by the importing code and data structures correspond. The cross-compiler cannot check for such correspondence.

The object to be imported or exported must be a variable declared at the outermost level of a library package specification or body.

The size of the object must be static. Thus, the type of the object must be one of:

- A scalar type (or subtype)
- An array subtype with static index constraints whose component size is static
- A nondiscriminated record type or subtype

Objects of a private or limited private type can be imported or exported only into the package that declares the type.

An imported object cannot have an initial value and thus cannot be:

- Declared with the keyword *constant*
- An access type
- A record type with discriminants
- A record type whose components have default initial expressions
- A record or array whose components contain access types or task types

In addition, the object must not be in a generic unit. The external name specified must be suitable as an identifier in the assembler.

The full syntax for the pragmas Import_Object and Export_Object is:

```
object_pragma       ::=  PRAGMA object_pragma_type
                            ( [ INTERNAL => ] identifier
                              [ , [ EXTERNAL => ] string_literal ] ) ;

object_pragma_type  ::=  IMPORT_OBJECT | EXPORT_OBJECT
```

Pragma Nickname

The pragma Nickname can be used to give a unique string name to a procedure or function in addition to its normal Ada name. This unique name can be used to distinguish among overloaded procedures or functions in the importing and exporting pragmas defined earlier.

The pragma Nickname must appear immediately following the declaration for which it is to provide a nickname. It has a single argument, the nickname, which must be a string constant. For example:

```
function Cat (L: Integer; R: String) return String;
pragma Nickname ("Int-Str-Cat");

function Cat (L: String; R: Integer) return String;
pragma Nickname ("Str-Int-Cat");

pragma Interface (Assembly, Cat);

pragma Import_Function (Internal  => Cat,
                        Nickname  => "Int-Str-Cat",
                        External  => "CAT$INT_STR_CONCAT",
                        Mechanism => (Value, Reference));

pragma Import_Function (Internal  => Cat,
                        Nickname  => "Str-Int-Cat",
                        External  => "CAT$STR_INT_CONCAT",
                        Mechanism => (Reference, Value));
```

Pragma Suppress_All

This pragma is equivalent to the following sequence of pragmas:

```
pragma Suppress (Access_Check);
pragma Suppress (Discriminant_Check);
pragma Suppress (Division_Check);
pragma Suppress (Elaboration_Check);
pragma Suppress (Index_Check);
pragma Suppress (Length_Check);
pragma Suppress (Overflow_Check);
pragma Suppress (Range_Check);
pragma Suppress (Storage_Check);
```

Pragma Suppress_All allows no name parameter, and it has no effect in a package specification. See LRM 11.7.3.

Note that, like pragma Suppress, pragma Suppress_All does not prevent the raising of certain exceptions. For example, numeric overflow or dividing by zero is detected by the hardware, which results in the predefined exception Numeric_Error. Refer to Chapter 7, "Runtime Organization," for more information.

Pragma Suppress_All must appear immediately within a declarative part.

PREDEFINED LANGUAGE PRAGMAS (LRM ANNEX B)

The following table details the effects of predefined language pragmas.

Predefined Pragmas

Pragma	Effect
Elaborate	As given in Annex B of the LRM.
Inline	As given in Annex B of the LRM, subject to the setting of the switch Inlining_Level.
Interface	Used in conjunction with pragmas Import_Procedure and Import_Function.
List	As given in Annex B of the LRM; evident only when the compile command is used.
Memory_Size	Has no effect.
Optimize	As given in Annex B of the LRM.
Pack	Removes gaps in storage, minimizing space with possible increase in access time. See section on size of objects.
Page	As given in Annex B of the LRM; only evident when the compile command is used.
Priority	As given in Annex B of the LRM.

Pragma	Effect
Shared	As given in Annex B of the LRM. Has an effect only for integer, enumeration, access, and fixed types.
Storage_Unit	Has no effect.
Suppress	As given in Annex B of the LRM.
System_Name	Has no effect.

IMPLEMENTATION-DEPENDENT ATTRIBUTES

The implementation-dependent attributes are as follows:

'Compiler_Version — For a name N, N'Compiler_Version is a compile-time value, a 16-character uninterpreted string that designates the compiler version used to code this Ada entity. The entity can be a program unit (package, subprogram, task, or generic), an object (variable, constant, named number, or parameter), a type or subtype (but *not* an incomplete type), or an exception. This attribute can be used for runtime detection of incompatibilities in data representation. See also 'Target_Key.

'Dope_Address — For an unconstrained array object A, A'Dope_Address is the address of the dope vector. The value is of type System.Address. This can be used for retrieving information about the object, as when reconstructing the array. See also 'Dope_Size.

'Dope_Size — For an unconstrained array object A, A'Dope_Size is the size in bits of the dope vector. The value is of type Universal_Integer. This can be used for retrieving information about the object, as when reconstructing the array. See also 'Dope_Address.

'Entry_Number — For a task entry or generic formal subprogram E, E'Entry_Number identifies the entity with a universal-integer value. This may be used by the runtime system to indicate that the entity corresponds to a process in the target and therefore requires an IPC queue.

'Mechanism — For a subprogram S with formal parameter P or the result of a function F, S'Mechanism(P) or F'Mechanism is a universal-integer value that designates the parameter-passing or function-return mechanism. This may be used by the IPC message-handling facilities to manipulate the data passed.

Currently, parameter-passing values and their meanings include:

1	parameter value is on the stack
2	parameter address is on the stack
3	parameter and dope vector address are on the stack
4	parameter address and 'Constrained datum are on the stack

Currently, function-return values and their meanings include:

11	result is returned in registers
15	address for the array result is on the stack
16	address for the result is in R0, result is on the stack
17	address for the record result is on the stack
18	address for the result is in R0, size in R1

Some details of parameter passing may change with new releases of the cross-compiler. See the release note for additional information.

'Target_Key — For a name N, N'Target_Key is a compile-time value, a 32-character uninterpreted string that designates the cross-compiler (that is, the target key) in effect when this entity was coded. This can be used for runtime detection of incompatibilities in data representation. See also 'Compiler_Version.

'Type_Key — For a type mark T declared in a subsystem, T'Type_Key is a unique 32-character uninterpreted string. This can be used for runtime type consistency checking of message data.

PACKAGE STANDARD (LRM ANNEX C)

Package Standard defines all the predefined identifiers in the language.

```
package Standard is

    type *Universal_Integer* is ...
    type *Universal_Real*    is ...
    type *Universal_Fixed*   is ...
    type Boolean             is (False, True);}

    type Integer             is range -2147483648 .. 2147483647;
    type Short_Short_Integer is range -128 .. 127;
    type Short_Integer       is range -32768 .. 32767;

    type Float       is digits 6  range -16#1.FFFF_FE# * 2.0 ** 127 ..
                                   16#1.FFFF_FE# * 2.0 ** 127;
    type Long_Float  is digits 15 range -16#1.FFFF_FFFF_FFFF_F# * 2.0 ** 1023
                                      .. 16#1.FFFF_FFFF_FFFF_F# * 2.0 ** 1023;

    type Duration    is delta 16#1.0# * 2.0 ** (-14)
                                  range -16#1.0# * 2.0 ** 17 ..
                                         16#1.FFFF_FFFC# * 2.0 ** 16;

    subtype Natural  is Integer range 0 .. 2147483647;
    subtype Positive is Integer range 1 .. 2147483647;

    type Character is ...

    type String is array (Positive range <>) of Character;
    pragma Pack (String);

    package Ascii is ...

    Constraint_Error : exception;
    Numeric_Error : exception;
    Storage_Error : exception;
    Tasking_Error : exception;
    Program_Error : exception;

end Standard;
```

The following table shows the default integer and floating-point types:

Supported Integer and Floating-Point Types

Ada Type Name	Size
Short_Short_Integer	8 bits
Short_Integer	16 bits
Integer	32 bits
Float	32 bits
Long_Float	64 bits

Fixed-point types are implemented using the smallest discrete type possible; it may be 8, 16, or 32 bits. Standard.Duration is 32 bits.

PACKAGE SYSTEM (LRM 13.7)

```
package System is

    type Address is private;

    type Name is (Mc68020_Os2000);

    System_Name : constant Name := Mc68020_Os2000;

    Storage_Unit : constant := 8;
    Memory_Size  : constant := +(2 ** 31) - 1;

    Min_Int : constant := -(2 ** 31);
    Max_Int : constant := +(2 ** 31) - 1;

    Max_Digits   : constant := 15;
    Max_Mantissa : constant := 31;
    Fine_Delta   : constant := 1.0 / (2.0 ** 31);
    Tick         : constant := 1.0 / (2.0 ** 13);

    subtype Priority is Integer range 0 .. 255;

    function To_Address (Value : Integer) return Address;
    function To_Integer (Value : Address) return Integer;

    function "+" (Left : Address; Right : Integer) return Address;
    function "+" (Left : Integer; Right : Address) return Address;
    function "-" (Left : Address; Right : Address) return Integer;
    function "-" (Left : Address; Right : Integer) return Address;
```

```
    function "<" (Left, Right : Address) return Boolean;
    function "<=" (Left, Right : Address) return Boolean;
    function ">" (Left, Right : Address) return Boolean;
    function ">=" (Left, Right : Address) return Boolean;
    --
    --  The functions above are unsigned in nature. Neither Numeric_Error
    --  nor Constraint_Error will ever be propagated by these functions.
    --
    --  Note that this implies:
    --
    --          To_Address (Integer'First) > To_Address (Integer'Last)
    --
    --  and that:
    --
    --          To_Address (0) < To_Address (-1)
    --
    --  Also, the unsigned range of Address includes values which are
    --  larger than those implied by Memory_Size.
    --

    Address_Zero : constant Address;

private
    . . .

end System;
```

REPRESENTATION CLAUSES AND CHANGES OF REPRESENTATION

The MC68020 CDF support for representation clauses is described in this section with reference to the relevant section of the LRM. Usage of a clause that is unsupported as specified in this section or usage contrary to LRM specification will cause a semantic error unless specifically noted. Further details on the effects on specific types of objects are given in the section "Size of Objects."

Length Clauses (LRM 13.2)

Length clauses are supported by the MC68020/OS-2000 CDF as follows:

- The value in a 'Size clause must be a positive static integer expression. 'Size clauses are supported for all scalar and composite types, including derived types, with the following restrictions:
  - — For all types the value of the size attribute must be greater than equal to the minimum size necessary to store the largest possible value of the type.
  - — For discrete types, the value of the size attribute must be less than or equal to 32.
  - — For fixed types, the value of the size attribute must be less than or equal to 32.

- — For float types, the size clause can only specify the size the type would have if there were no clause.
- — For access and task types, the value of the size attribute must be 32.
- — For composite types, a size specification must not imply compression of composite components. Such compression must have been explicitly requested using a length clause or pragma pack on the component type.

- 'Storage_Size clauses are supported for access and task types. The value given in a Storage_Size clause can be any integer expression, and it is not required to be static.
- 'Small clauses are supported for fixed-point types. The value given in a 'Small clause must be a nonzero static real number that cannot be greater than the delta of the base type.

Enumeration Representation Clauses (LRM 13.3)

Enumeration representation clauses are supported; the allowable values for an enumeration clause range from (Integer'First + 1) to Integer'Last.

Record Representation Clauses (LRM 13.4)

Both full and partial representation clauses are supported for both discriminated and undiscriminated records. Record component clauses are not allowed on:

- Array or record fields whose constraint involves a discriminant of the enclosing record
- Array or record fields whose constraint is not static

The **static_simple_expression** in the alignment clause part of a record representation clause (see LRM 13.4 (4)) must be a power of 2 with the following limits:

```
1 <= static_simple_expression <= 16
```

Implementation-Dependent Components

The LRM allows for the generation of names denoting implementation-dependent components in records. For the MC68020/OS-2000 CDF, there are no such names visible to the user.

Address Clauses (LRM 13.5)

Address clauses are not supported and will generate a semantic error if used.

Change of Representation (LRM 13.6)

Change of representation is supported wherever it is implied by support for representation specifications. In particular, type conversions between array types or record types may cause packing or unpacking to occur; conversions between related enumeration types with different representations may result in table lookup operations.

SIZE OF OBJECTS

This section describes the size of both scalar and composite objects. The first two subsections cover concepts of size that apply to all object types. The following subsections cover individual types. The size concepts are most important for the composite types.

Minimum, Default, Packed, and Unpacked Sizes

The following terms are used to describe the size of objects:

- Storage unit: Smallest addressable memory unit. The size of the storage unit in bits is given by the named number System.Storage_Unit. Since the MC68020 is byte-addressable, the size of the storage unit is 8.
- Minimum size for a type: The minimum number of bits required to store the largest value of the type. For example, the minimum size of a Boolean is 1.
- Packed size for a type: The size of a component used in an array or record when a pragma Pack is in effect. This is the same as the minimum size unless modified by a 'Size clause (see "Determination of Size" below).
- Default size for a type: The smallest number of bits required to store the largest value of the type *when stored in whole storage units*. For composite types, the default sizes are multiples of 8. The possible default sizes for scalar, access, and task types are given in the following table:

Default Sizes for Scalar, Access, and Task Types

Type	Sizes in Bits
Discrete	8, 16, 32
Fixed	8, 16, 32
Float	32, 64
Access	32
Task	32

- Unpacked size for a type: The size of a component used in an array or record when no pragma Pack is in effect. This is the same as the default size unless modified by a 'Size clause (see "Determination of Size" below).
- Maximum size: The largest allowable size for a *discrete* type. For the MC68020, the maximum size is 32.

Determination of Size

Top-level scalar and access objects are stored using their unpacked size (by top-level object we mean an object that is not a component of any array or record). Components of composite objects having neither pragma Pack nor a record representation clause are also stored using the unpacked size. Components of composite objects having pragma Pack are stored using the packed size. Fields of records having record representation clauses may be stored in any number of bits ranging from the minimum size to the default size of the field type. If a scalar or composite type component field is specified to be smaller than the default size, a filler field is introduced, and the data is left-justified. For further information, see the subsections on composite types.

'Size clauses on discrete types affect sizes by changing the packed and unpacked sizes. When there is no 'Size clause, the packed and unpacked sizes are the minimum and default sizes, respectively. 'Size clauses with values outside the minimum and maximum sizes cause a semantic error. Within that range, there are two cases depending on the value specified by the clause:

- *Value* <= Default Size: The packed size is set equal to *value*. The unpacked size is not affected.
- *Value* > Default Size: The packed size is set equal to *Value* and the unpacked size is set to the number of bits in the smallest number of storage units that will hold the packed size.

Size Examples for MC68020 Target

Type Declaration of the Example	Minimum Size	Default Size	Maximum Size
Integer	32	32	32
Boolean	1	8	32
Float	32	32	32
type Byte is range 0 .. 255	8	16	32
type Primary is (Red, Blue, Yellow)	2	8	32
type X is (Normal, Read_Error, Write_Error) for X use (7, 15, 31)	5	8	32
type Ary is array (1 .. 100) of Boolean	100	800	n/a

Integer Types

An integer type with a range constraint has a default size the same as that of the smallest integer type defined in package Standard that will hold its range. For example, consider the following type declaration:

```
type Byte is range 0..255;
```

Type Byte will have a minimum size of 8 and a default size of 16. It has a default size of 16 because the smallest type from which Byte can be derived is Short_Integer. (Short_Short_Integer, which has a size of 8, does not include values greater than 127.)

The 'Size clause is supported for integer types and derived types. The effect of a 'Size clause on minimum size is shown in the following example. Consider:

```
type Byte is range 0 .. 255;
for Byte'Size use n;
```

where *n* is a static integer expression. The following table shows the effect of *n* on the packed and unpacked sizes.

Example of Effect of 'Size Clauses

'Size Clause	Packed Size	Unpacked Size
No 'Size clause	8	16
Use 8	8	16
Use 12	12	16
Use 16	16	16

Integer types of range 0..255 may be stored in a byte, rather than a word by using 'Size. This may generate extra code to perform type conversions, as shown in the following example.

```
package Blat is
  type Byte is range 0..255;
   procedure X (B: Byte);
end;

type Zap is new Byte;
For Zap'Size use 8;
```

This will generate conversion code every time type Zap is used as a parameter to procedure X. Procedure X itself will not contain the extra code, nor will two procedure X's be generated.

Enumeration Types

For an enumeration type with n elements, the default internal integer representation range is 0 .. n–1. The maximum number of elements that may be declared for any one enumeration type depends on the total number of characters in the images of the enumeration literals. Let L be the total number of characters of the n elements. Then L and n must satisfy the following inequality: $2n + 4 + L < 2^{16}$.

Enumeration and 'Size clauses are permitted on derived types. However, this may generate additional code when parent/derived types are converted to each other.

For predefined type Character, the value returned by the 'Size attribute is 8, and the minimum size is 8. User-defined Character types behave like ordinary enumeration types and may have a minimum size of less than 8.

The 'Size clause is supported for enumeration types and derived types. The effect of a 'Size clause on representation is shown in the following example. Consider:

```
type Response is (No, Maybe, Yes);
for Response'Size use n;
```

where n is a static integer expression. The following table lists the packed and unpacked sizes for different values of n.

Example

'Size Clause	Packed Size	Unpacked Size
No 'Size clause	2	8
Use 4	4	8
Use 12	12	16
Use 16	16	16
Use 20	20	32
Use 32	32	32

Floating-Point Types

The internal representations for floating-point types are the 32-bit and 64-bit floating-point representations as outlined by the MC68020 architecture.

Fixed-Point Types

Fixed-point numbers are represented internally as integers. The integer representation is computed by scaling (dividing) the fixed-point number by the actual small implied by the fixed-point type declaration. The values that are exactly representable are those that are precise multiples of the actual small; numbers between those values are represented by the closest exact multiple. For example, in the declaration:

```
type fix is delta 0.01 range -10.0 .. 10.0;
```

the integer value used to represent the lower bound of the type is –10.0 / (1/128), or –1280, since the actual small is 1/128. In the example:

```
type fix is delta 0.01 range -10.6 .. 10.6;
```

the integer value used to represent the lower bound of the type is –1357, which is the closest exact multiple of the actual small.

The size of the representation is 32, 16, or 8 bits; the compiler chooses the smallest of these that can represent all of the safe numbers of the fixed-point type.

Access Types and Task Types

Access and task objects have a size of 32 bits. The 'Storage_Size length clause is allowed for access and task types. The value given in a Storage_Size clause may be any integer expression, and it is not required to be static. Static expressions larger than Integer'Last will generate compilation warnings; however, a Numeric_Error exception will be raised at run time. For access types, a 'Storage_Size clause is used to specify the size of the access type's collection. If a 'Storage_Size clause has been applied to an access type, the collection is nonextensible. For task types, the clause determines the stack size.

A value (either static or not) of 0 is allowed; in this case, no collection or task stack space will be allocated, and Storage_Error will be raised at run time if any attempt is made to allocate or deallocate from the collection or activate the task. Negative values are also allowed by the CDF; however, this will generate a Storage_Error exception when the type is elaborated even if no attempt is made to allocate or deallocate objects belonging to the collection.

Composite Types

The size of a composite type depends on whether or not it is packed. Whether it is packed or not depends the presence or absence of

- pragma Pack,

- pragma Optimize,
- in certain situations, a 'Size clause,

as specified in subsection on representation selection.

Other factors affecting size are the presence of:

- A record representation clause
- Alignment filler
- Tail filler

These factors, which mostly affect records, are dealt with first in the following subsections. Once their effects are understood, the differences between packed and unpacked representations are fairly simple. Later subsections deal with packed and unpacked representation and pragma Pack.

Using a Record Representation Clause

If any components or parts of components are covered by a record representation clause, then that clause controls, whether the composite type is packed or not.

If a record representation clause is used, some fields of a record may not be determined by that clause. Such fields may be influenced by packing. Also, these fields are placed after all those that are specified by the clause. In particular, for a discriminant not governed by a clause, the discriminant field is placed after all the governed fields. Since the discriminant field must then be placed after the largest possible field, there will be no space savings gained by using the constrained subtype of the discriminated record.

Alignment Filler

Alignment filler may be present in a composite type when it is a record or is an array containing records. In the absence of a record representation clause, the compiler reorders record fields in an attempt to maintain alignment and reduce the need for alignment filler fields. Nonetheless, sometimes it is necessary to introduce alignment filler fields to maintain alignment. For example, in an unpacked array of records consisting of a character and an integer, the integers would be aligned on longword boundaries by introducing an alignment filler of three bytes between array components.

Tail Filler

The last storage unit of the composite type may contain some unused bits; these bits, called *tail filler*, are zeroed when an object is elaborated so that block comparisons can be performed. Tail filler does *not* contribute to the size of the composite type as computed by the 'Size attribute.

The Choice between Packed and Unpacked Representation

The criteria for selecting a representation are as follows:

- If there is a pragma Pack on the composite type, then the packed representation is used.
- If there is a legal 'Size clause on the composite type which cannot be satisfied by the unpacked representation, then the packed representation is used.
- If there would be alignment filler in the unpacked representation, then the packed representation is used *unless* there is a `pragma Optimize(Time)` in effect at the point of the

composite type definition.

- In other all other cases, the unpacked representation is used.

Unpacked Composite Types

When a composite type is unpacked, each component is stored in the same size it would have as a top-level object. For scalars, that is its unpacked size. Components covered by a record representation clause obey that clause. For components of composite type, any packing within the component will be retained.

Packed Composite Types

When a composite type is packed, scalar components are stored in packed size. Tail filler between compo- nents is eliminated; for arrays, alignment filler within components is eliminated; for records, it is not.

Change of Representation for Packed Composite Types

Change of representation for packed composite types may cause extra code to be generated to do packing and unpacking conversion. For example:

```
type A is array (1..10) of Boolean;  -- Size of 80 bits
type B is new A;
pragma Pack (B);                     -- Size of 10 bits

X : B;
Y : A;
X := B(Y);    -- Extra code will be generated here
              -- to convert from type B to type A
```

OTHER IMPLEMENTATION-DEPENDENT FEATURES

Machine Code (LRM 13.8)

Machine-code insertions are not supported at this time.

Unchecked Storage Deallocation (LRM 13.10.1)

Unchecked storage deallocation is implemented by the Unchecked_Deallocation generic function defined by the LRM. This procedure can be instantiated with an object type and its access type, resulting in a procedure that deallocates the object's storage. Objects of any type may be deallocated.

The storage reserved for the entire collection associated with an access type is reclaimed when the program exits the scope in which the access type is declared. Placing an access type declaration within a block can be a useful implementation strategy when conservation of memory is necessary.

Erroneous use of dangling references may be detected in certain cases. When detected, the Storage_Error exception is raised. Deallocation of objects that were not created through allocation (that is, through Unchecked_Conversion) may also be detected in certain cases, also raising Storage_Error.

Unchecked Type Conversion (LRM 13.10.2)

Unchecked conversion is implemented by the Unchecked_Conversion generic function defined by the LRM. This function can be instantiated with Source and Target types, resulting in a function that converts source data values into target data values.

Unchecked conversion moves storage units from the source object to the target object sequentially, starting with the lowest address. Transfer continues until the source object is exhausted or the target object runs out of room. If the target is larger than the source, the remaining bits are undefined.

Restrictions on Unchecked Type Conversion

- The target type of an unchecked conversion cannot be an unconstrained array type or an unconstrained discriminated type without default discriminants.
- Internal consistency among components of the target type is not guaranteed. Discriminant components may contain illegal values or be inconsistent with the use of those discriminants elsewhere in the type representation.

CHARACTERISTICS OF I/O PACKAGES

This section specifies the implementation-dependent characteristics of the I/O packages Sequential_Io, Direct_Io, Text_Io, and Io_Exceptions. These packages are located in the library !Targets.Mc68020_Os2000.Io. Nondependent characteristics are as given in Chapter 14 of the LRM. Each section below cites the relevant section in Chapter 14.

The package Os2000_Io is provided as an interface to many of the OS-2000 I/O system calls. Os2000_Io is in the library !Targets.Mc68020_Os2000.Target_Interface.

Package Low_Level_Io is not provided for the Mc68020_Os2000 target.

External Files and File Objects (LRM 14.1)

An external file is identified by a Name and further characterized by a Form parameter. The allowable strings for the Name are the legal filenames and full or relative path lists accepted by OS-2000. See the *OS-9/68000 Operating System User's Manual* for details. The recognized values for the Form parameter string are those containing the identifiers Non_Sharable or Sharable. All other elements of the Form string are ignored. These values of the Form parameter determine the setting of the Os2000 Single_User bit when opening or creating the external file. If the Form is Sharable, then the Single_User bit is true, if the Form is Non_Sharable, then the Single_User bit is false. The default values for the Form are Sharable for In mode files and Non_Sharable for Inout and Out mode files. If a Form parameter contains both identifiers Non_Sharable and

Sharable, then Form will be Non_Sharable.

If a main program completes without closing some Text_Io file, any data output to the file after the last line or page terminator will not be included in the associated external file.

Input and output are erroneous for access types.

Sequential and Direct Files

This section deals with implementation-dependent features associated with the packages Sequential_Io and Direct_Io and the file types *sequential access* and *direct access*.

File Management (LRM 14.2.1)

The Use_Error exception is raised in the following situations:

- By procedure Create if an external file with the specified Name already exists.
- By procedure Open if the executing process does not have correct access rights for the external file.
- By procedure Open if another process has already opened the external file for nonsharable use.
- By either procedure Create or Open if accessing the external file would exceed the OS-2000 limit on the number of open files for the executing process.
- By procedure Open if the external file is currently opened by another process with mode Out or Inout.

Sequential Input/Output (LRM 14.2.2)

For the Read procedure of Sequential_Io, the Data_Error exception is raised only when the size of the data read from the file is greater than the size of the *out* parameter Item.

Direct Input/Output (LRM 14.2.4)

Package Direct_Io may not be instantiated with any type that is either an unconstrained array type or a discriminated record type without default discriminants. A semantic error is reported when attempting to install any unit that contains an instantiation in which the actual type is such a forbidden type.

For the Read procedure of Direct_Io, there is no check performed to ensure that the data read from the file can be interpreted as a value of the Element_Type.

Specification of Package Direct_Io (LRM 14.2.5)

The declaration of the type Count in package Direct_Io is:

```
type Count is new Integer range 0 .. Integer'Last / Element_Type'Size;
```

where Element_Type is the generic formal type parameter.

Text Input/Output (LRM 14.3)

The Text_Io default input and output files are associated with the OS-2000 standard input and standard output paths, respectively. If a program is initiated without any of the standard input, standard output, or standard error output, then those paths are opened for device/nil. The terminators used by Text_Io are Ascii.Cr for the line terminator, and the sequence Ascii.Ff, Ascii.Cr for the page terminator, except for terminators at the end of file, which are implicit and not represented by any characters.

File Management (LRM 14.3.1)

The Text_Io function Name raises Use_Error when called with either Text_Io.Standard_Input or Text_Io.Standard_Output as the actual parameter. This implementation for Name was chosen because OS-2000 does not provide a uniform mechanism for obtaining a string name for the external file associated with a path.

Specification of Package Text_Io (LRM 14.3.10)

The declaration of the type Count in Text_Io is:

```
type Count is range 0 .. 1_000_000_000;
```

The declaration of the subtype Field in Text_Io is:

```
subtype Field is Integer range 0 .. Integer'Last;
```

Exceptions in I/O (LRM 14.4)

The exceptions raised in input/output operations are:

- The exceptions as specified in LRM 14.4.
- The Use_Error exception as specified in the two file management subsections above.
- The Device_Error exception, raised for any input/output operation that performs an OS-2000 I/O system call returning as status the error code E$Read or E$Write.
- The Device_Error exception, raised if the status returned from any OS-2000 I/O call is not among the error codes E$Pnnf, E$Bpnam, E$Bmode, E$Eof, E$Pthful, E$Fna, E$Share, E$Bpnum, ECef, ERead, or E$Write.